SketchTree: Approximate Tree Pattern Counts over Streaming Labeled Trees *

Praveen Rao
Department of Computer Science
University of Arizona, Tucson, AZ 85721
rpraveen@cs.arizona.edu

Bongki Moon
Department of Computer Science
University of Arizona, Tucson, AZ 85721
bkmoon@cs.arizona.edu

Abstract

*In recent years, there has been a rising interest in developing online approximation algorithms for data streams. Some of the key challenges are posed by the fact that streaming data can be read only once in a fixed order of arrival and only a limited amount of memory is available for storage. In this paper, we address the problem of approximately counting tree patterns over a stream of labeled trees (*e.g.*, XML documents). We propose a new approximation algorithm called* `SketchTree` *that computes a synopsis of the stream in a single pass by processing each tree only once. Using a limited amount of memory,* `SketchTree` *provides approximate answers for both ordered and unordered tree pattern counts. Furthermore, we discuss a class of count queries that can be handled by* `SketchTree` *and their utility. We provide theoretical analyses to show that our algorithm has provably strong guarantees on the error bounds. Experiments on real datasets demonstrate that* `SketchTree` *can indeed estimate tree pattern counts within 10-15% relative error with high confidence under various situations.*

1. Introduction

In recent years, the area of data stream processing has received much attention with key focus on developing online algorithms using a limited amount of memory. The algorithms are single-pass in nature in that every stream element is examined only once. Internet service providers, e-commerce companies and applications such as network monitoring and sensor data collection, constantly gather and analyze a large amount of data to detect trends and/or anomalies in their systems. The volume of data generated by these applications obviates any traditional indexing and storing techniques. As a result, such applications necessitate efficient algorithms that can provide statistics or summaries on the data using a limited amount of memory.

Recent research in data streaming has focused on developing approximation algorithms with strong guarantees on the error bound. A popular approach has been to compute online synopsis on data streams in a limited space and use the synopsis for approximate query processing. Some of the key challenges that arise in the streaming environment are (a) to develop a synopsis data structure that requires space *logarithmic* or *poly-logarithmic* in the length of the stream and (b) to compute the synopsis in a single pass over the stream by incurring a small per-element processing cost. Several theoretical and experimental studies have been conducted such as online computation of frequency moments [3], join size estimation [2, 11], online quantile computation [13, 14], and tracking frequent elements [9, 21].

The utility of tree structures spans across many areas such as modeling XML documents, representing phylogenies in biological applications, networks, web log analysis and so on. Today, the extensible markup language XML is a popular standard for information representation and exchange on the Internet [5]. Many emerging applications such as personalized news, stock quotes, and price alerts have become popular over the Internet. The rich data and query semantics provided by XML has triggered several research attempts to build selective information dissemination systems [4, 18], content-based routing systems [10, 27] and XQuery processors [17, 20] for streaming XML data. There is a growing interest in developing software systems for efficiently processing XML streams.

While *finding all occurrences* of a query pattern in tree structured data such as XML documents is one of the core operations on stored data (*e.g.*, XISS [19], TwigStack [7], TSGeneric$^+$ [16], PRIX [26]), it may not always be necessary to do so for the purpose of analyzing trends in the online activities. Rather it may be desired to *count* all matching occurrences from streaming data in a real-time fashion without consuming too much computing resource.

Problem Description

In this paper, we propose a new algorithm called `SketchTree` for approximately counting all matching occurrences of a tree pattern in a stream of labeled trees. Consider the problem of counting the number of matches of a query pattern Q in a stream of trees processed from left-to-right shown in Figure 1. The query Q contains a root node A with B and C as its children. Suppose we want to count those *ordered matches* for Q where B precedes C in the data. Tree T_1 has two matches and tree T_3 has one match. Suppose we want to count

* This work was sponsored in part by National Science Foundation CAREER Award (IIS-9876037), NSF Grant IIS-0100436, and NSF Research Infrastructure program EIA-0080123. It was also supported by the Prop 301 Fund from the State of Arizona. The authors assume all responsibility for the contents of the paper.

Proceedings of the 22nd Inter. Conference on Data Engineering (ICDE'2006)
Atlanta, GA, April, 2006.

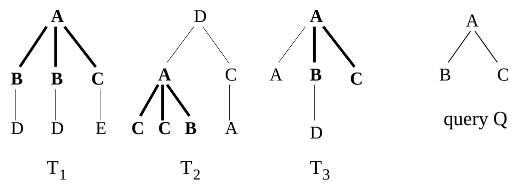


Figure 1. A Stream of Labeled Trees and a Query Pattern

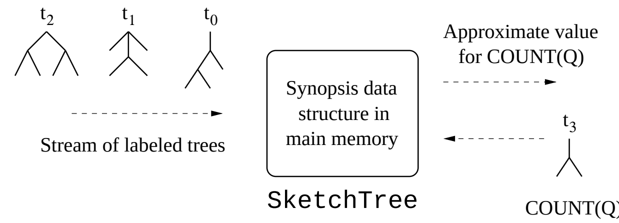


Figure 2. Streaming Model for `SketchTree`

those *unordered matches* for Q with no ordering constraint between B and C, then tree T_2 has two matches. (The matching nodes in the trees are shown in bold and the matching edges are drawn thick.) We shall use the above semantics for query pattern matching in our work, which is slightly different from the XPath query semantics for XML. The details of the query semantics of `SketchTree` are discussed in Section 2.1,

To the best of our knowledge, this work is the first attempt to address the problem of counting tree pattern matches over streaming labeled trees such as XML documents using a limited amount of memory. Formally, we state the tree pattern counting problem as follows.

> *Given a stream of labeled trees that are looked at only once in the fixed order in which they arrive, count all matching occurrences of a tree pattern in the stream so far.*

Note that this problem is fundamentally different from the problem of filtering for selective information dissemination [4, 18], where user profiles are represented as standing XPath queries. In the aforementioned tree pattern counting problem, there exist *no* standing queries to begin with, and *any* tree pattern can be thrown as a query at *any* moment in time during stream processing.

Motivations and Contributions

To motivate why an approximate counting strategy may be more desirable than tracking counts accurately, let us consider a stream of ordered labeled trees with node labels chosen from a finite symbol set Σ. In order to accurately count the number of occurrences for any tree pattern of n nodes, it is necessary to maintain a counter for each of all possible tree patterns of n nodes. If all node labels in the trees are ignored, then the total number of distinct ordered unlabeled tree patterns of n nodes is given by $\frac{1}{n}\times \binom{2n-2}{n-1}$ [12]. Since each node in an unlabeled tree pattern can be assigned any one of the labels in Σ, the number of counters required in the worst case, to count all possible labeled tree patterns of n nodes, is $\frac{1}{n}\times \binom{2n-2}{n-1}$ $\times|\Sigma|^n$ in the worst case. In the worst case, each counter requires $lg(m)$ bits, where m is the total number of tree patterns in the stream.

Therefore, the memory requirement may be impractically too high for most realistic applications with non-trivial alphabet size $|\Sigma|$ and tree size n. For applications that only need approximate counts with provable guarantees on error bounds, it would be useful to provide a method that approximately counts all matching occurrences of any tree pattern using a substantially smaller amount of memory than that required for accurate counts.

The main contributions of this paper are summarized as follows.

- We propose a new online approximation algorithm `SketchTree` for counting tree patterns over a stream of labeled trees using a limited amount of memory with provably strong error bounds.
- We show that `SketchTree` can estimate counts for a class of queries that includes both ordered and unordered tree patterns.
- To reduce the memory requirement of `SketchTree` for guaranteeing a certain level of accuracy, we propose two strategies that aim at reducing the self-join size of a stream.
- We have developed an intuitive algorithm $EnumTree$ for efficiently enumerating all the tree patterns in a tree with at most k edges each.
- We have validated the effectiveness of `SketchTree` using two real datasets with different characteristics.

The rest of this paper is organized as follows. In Section 2, we provide an overview of the streaming model for labeled trees and basic techniques. In Section 3, we present the synopsis data structure used by `SketchTree` with theoretical analyses. Section 4 discusses a class of count queries supported by `SketchTree` with some use cases. Section 5 discusses strategies to improve `SketchTree`'s processing cost and estimation accuracy. In Section 6, we present some extensions to `SketchTree` followed by experimental results in Section 7. Lastly, Section 8 summarizes the contributions of this paper.

2. Streaming Model and Basic Techniques

We begin with a brief description of the streaming model used by `SketchTree`. As is illustrated in Figure 2, a synopsis data structure is continuously updated, while each of labeled trees (*e.g.*, XML documents) are processed. At the end of time t_2, three trees have been processed by the system. In the figure, a count query for Q is issued at time t_3 and the system returns an approximate answer.

2.1. Query Semantics

Query patterns supported by `SketchTree` are labeled trees. The edges in a query Q denote a parent-child relationship between nodes (similar to '/' axis in XPath). In this paper, we restrict Q to contain only equality predicates. A value in a predicate is treated as a node label. For the stream processed so far, we use $COUNT(Q)$ to denote the number of all occurrences of Q in the stream, where the matches are *unordered* in nature. In addition, we use $COUNT_{ord}(Q)$ to denote the number of all occurrences of Q, where the matches are *ordered* in nature. `SketchTree` reports approximate answers for such queries.

It should be noted that our query semantics is slightly different from XPath, in the sense that `SketchTree` considers all occurrences of a query pattern whilst XPath considers all occurrences of a target element in an XPath query. Suppose we want to process $COUNT(Q)$ for the trees shown in Figure 1. Using our query semantics, $COUNT(Q) = 5$. On the other hand, using XPath semantics, $COUNT(\texttt{//A[B]/C}) = 4$.

In the streaming scenario studied in this paper, we assume that there exists no structural summary such as a schema for the input data. However, if a structural summary is available or can be constructed online for the data in limited space, then `SketchTree` can be extended to efficiently process queries with ancestor-descendant relationship between nodes (similar to '//' in XPath) and wildcard nodes (similar to '*' in XPath). We defer the discussion of these extensions until Section 6.

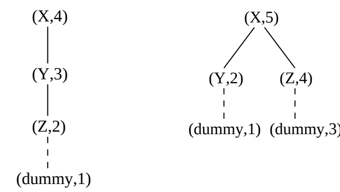
2.2. Counting Parent-Child Node Pairs

Suppose we want to count the number of occurrences of any parent-child node pair in a stream of labeled trees. Let Σ denote the set of all possible node labels. A naive counting algorithm would require one counter for each possible ordered pair of labels to count all possible parent-child node pairs. Thus a total of $|\Sigma|^2$ counters are required. Initially, each counter is set to zero. For a new tree in the input stream, all the parent-child pairs are determined and their corresponding counters are updated. At any moment, the result for $COUNT(\cdot)$ can be obtained from an appropriate counter.

Alternatively, we can process the trees in the following way. Let $hash(X)$ denote a function that returns a unique number for any given node label X. Then any pair (X,Y) of parent-child nodes can be represented by an ordered pair $(hash(X), hash(Y))$. Without loss of generality, we will initially assume that each node label hashes to a unique number. Later in Section 6, we will describe how to overcome this assumption.

Using the notion of *pairing functions*, any 2-tuple can be uniquely mapped to a natural number [15]. Pairing functions provide a one-to-one mapping between an ordered pair of non-negative integers and a single non-negative integer. Tuples with more than two elements can also be mapped to single numbers by applying pairing functions inductively as follows.

$$
\begin{aligned}
PF_2(x,y) &= \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y) \\
PF_3(x,y,z) &= PF_2(PF_2(x,y),z)
\end{aligned}
$$



(a) Pattern T_1 (b) Pattern T_2

Figure 3. Example Tree Patterns

We shall use the notation $PF(\cdot)$ to denote a family of pairing functions for *k-tuples*. By applying pairing functions on the ordered pairs of node labels, a stream of labeled trees can be mapped to a stream of integers (or one-dimensional points). Existing techniques for computing point estimates with limited memory (*e.g.*, AMS sketches [3], COUNT sketches [8]) can be used to estimate the number of occurrences of any parent-child pair in the stream.

2.3. Counting Tree Patterns

In this section, we describe how `SketchTree` can estimate tree pattern counts. The key idea of our approach is to map tree patterns into Prüfer sequences and eventually map these sequences into one-dimensional integers. The use of Prüfer sequence representation for XML document trees was first proposed in the PRIX system [26] for indexing and querying XML.

Prüfer sequences provide a one-to-one mapping between a labeled tree and a sequence. The algorithm to construct a sequence from a labeled tree deletes successively the leaf node with the smallest label and notes down the parent node of the deleted one. This process continues until only one node is left. The ordered sequence of noted nodes becomes the Prüfer sequence of the tree. As in the PRIX system, we shall construct a Prüfer sequence of length $n-1$ for a labeled tree T_n of n nodes by continuing the deletion of nodes till only one node is left. Note that the time complexity of constructing a Prüfer sequence is linear in the number of tree nodes [26].

The nodes of a labeled tree are first assigned postorder numbers. As in the PRIX system, the Prüfer sequence can be constructed by treating the postorder numbers as unique labels for the node removal method described above. Two sequences are constructed: (1) *NPS (Numbered Prüfer sequence)* consisting entirely of postorder numbers and (2) *LPS (Labeled Prüfer sequence)* obtained by replacing each number in the NPS by its corresponding label. For the purpose of tree pattern counting, we produce *extended Prüfer* sequences by adding a dummy child node to each of the leaf nodes of a labeled tree before applying the sequence construction. The Prüfer sequence of the extended tree contains the leaf labels of the original tree. The LPS and NPS of the extended tree together contain complete information needed to reconstruct the original labeled tree [26].

Example 1 *We shall convert the tree patterns in Figure 3 into sequences. Each node has a label and a postorder number. The nodes of the tree patterns that appear in the data tree are connected by solid edges. The original leaf nodes in* T_1 *and*

T_2 are extended by adding dummy nodes (connected by dotted edges). All the nodes including the dummy nodes are numbered in postorder. T_1 can be uniquely represented by its LPS(T_1) = Z Y X, and NPS(T_1) = 2 3 4. T_2 can be uniquely represented by its LPS(T_2) = Y X Z X, and NPS(T_2) = 2 5 4 5.

We first deal with estimating $COUNT_{ord}(\cdot)$ queries over a stream of labeled trees. Later in Section 3.3, we extend `SketchTree` to estimate $COUNT(\cdot)$ queries (unordered matches) with provable error guarantees.

In the `SketchTree` algorithm, Prüfer sequence representation is adopted for both the data trees and query tree patterns. A brief outline of the `SketchTree` algorithm is presented as follows. Let us assume an algorithm $EnumTree(T, k)$ that enumerates all ordered tree patterns in T with at most k edges each. (The details of $EnumTree(\cdot)$ will be discussed in Section 5.1.) When a new data tree arrives in the stream, `SketchTree` enumerates all the tree patterns in this tree with one to k edges using $EnumTree$. For each tree pattern generated from the tree, the (extended) LPS and NPS for the pattern are constructed as in Example 1. By applying a pairing function to each pair of LPS and NPS, the stream of trees are mapped into a stream of one-dimensional integer values. Since the LPS and NPS together uniquely identify a tree pattern, every distinct tree pattern is mapped to a distinct integer using $PF(\cdot)$. As a result, the problem of estimating tree pattern counts is reduced to that of approximately estimating the frequency of one-dimensional points in a stream.

Example 2 *Suppose the patterns T_1 and T_2 in Figure 3 are generated by $EnumTree(\cdot)$. For T_1, LPS(T_1) = Z Y X, and NPS(T_1) = 2 3 4. Then the sequences can be mapped to one-dimensional values as follows. We compute ρ_1 = PF(hash(Z), hash(Y), hash(X), 2, 3, 4) by treating all the elements in the LPS and NPS as part of one long tuple. Similarly for T_2, we compute ρ_2 = PF(hash(Y), hash(X), hash(Z), hash(X), 2, 5, 4, 5).*

When required, we shall use '.' to denote the concatenation of a LPS L and a NPS N. Then $PF(L.N)$ denotes the one-dimensional mapping. The pairing function provides a one-to-one mapping if all the tuples are of the same length. If not, each tuple should be padded to the size of the largest tuple before being mapped to a value. For ease of explanation, we shall assume that this padding functionality is incorporated in the pairing function. It is evident that the range of $PF(\cdot)$ grows rapidly with increase in the length of the tuple and value of the tuple elements. If the range of $PF(\cdot)$ becomes too large to be represented in fixed length words (*e.g.,* 32 or 64 bit words), we use an alternate strategy that computes residues using irreducible polynomials of high degrees. Note that this strategy does not require padding the sequences. In Section 6, we explain the process of mapping sequences using irreducible polynomials. Until then, we shall continue to use $PF(\cdot)$ for the mapping process.

3. Synopsis Data Structure

The synopsis data structure maintained by `SketchTree` for a stream of labeled trees is based on AMS sketches [3]. In their seminal work, *Alon, Matias and Szegedy* (hence the name AMS sketches) proposed the use of randomized linear projection of the frequency vector of the values in a stream. The process of computing a randomized linear projection X of the frequency vector of a stream S can be summarized as follows [3].

- Let $dom(S) = \{1, 2, ..., n\}$ be the domain of S of size n. Select at random a family of *four-wise* independent binary random variables $\xi_i = \{-1, +1\}$ for each $i \in dom(S)$. Note that $P(\xi_i = -1) = P(\xi_i = +1) = \frac{1}{2}$ and $E(\xi_i) = 0$. By four-wise independence, we mean that for any 4-tuple of ξ_i's and any 4-tuple of $\{-1, +1\}$ values, the probability that these two 4-tuples match is $\frac{1}{16}$.
- Compute $X = \sum_{i=1}^{n} f_i \xi_i$ for the values in S, where f_i is the frequency of the value i in S. This can be done online as follows. Initialize $X = 0$. Each time a value i occurs in S, simply add ξ_i to X.

The four-wise independent binary random variables can be generated by constructing parity check matrices of the binary BCH codes [3]. Each sketch requires memory in the order of log of the domain size and the log of the length of the stream. A useful property of AMS sketches is that deleting values from a stream is easy. A value i can be deleted from the stream S by subtracting ξ_i from X.

Our choice of AMS sketches for `SketchTree` was influenced by the fact that these sketches have interesting mathematical properties that allow us to construct *unbiased estimators* in an intuitive way for a class of count queries over tree structured data. Furthermore, provable bounds for the approximation error can be computed for these queries in a *methodical way*. In the following sections, we present theoretical analyses for estimating a class of count queries using `SketchTree`. Our style of analysis is similar to that of Alon *et al.* [3] in the sense that first an unbiased estimator is constructed and then its variance is computed. This is followed by the application of Chebyshev's Inequality and Chernoff bounds [22] to formulate theorems regarding the accuracy of the estimators. We shall use the terms 'frequency' and 'count' interchangeably in the following discussions.

3.1. Estimating the Frequency of a Tree Pattern

We shall describe how `SketchTree` can estimate the frequency of a tree pattern Q (*i.e.,* $COUNT_{ord}(Q)$). Let $dom(S)$ denote the range of the pairing function $PF(\cdot)$ used to map tree patterns into a stream of one-dimensional values S. Then a sketch X can be computed for S as explained before. Let q denote the $PF(\cdot)$ value for query Q. It is straightforward to show that $E(\xi_q \cdot X)$ is an unbiased estimator of $COUNT_{ord}(Q)$. Note that $E(\xi_i^2) = 1$ and $E(\xi_i \xi_j) = 0$ if $i \neq j$. By linearity of expectation,

$$E(\xi_q \cdot X) = E(\xi_q \cdot (\xi_1 f_1 + \cdots + \xi_n f_n)) = E(\xi_q^2 f_q) = f_q. \quad (1)$$

Let $SJ(S)$ denote the self-join size of stream S. By applying the standard formula for variance, it can be shown that $Var[\xi_q \cdot X] \leq SJ(S)$ [25].

The accuracy of estimation can be improved by applying the standard *boosting technique* [3] that maintains $s_1 \times s_2$ independent and identically distributed (iid) instances of X (*i.e.,* X_{ij}), where s_1 and s_2 are constants. We compute s_2 random variables Y_1, Y_2,..., Y_{s_2} as follows. Each Y_i is the average of s_1 iid instances of $\xi_q X$. The median of Y_1, Y_2,..., Y_{s_2} is an improved estimate of $COUNT_{ord}(Q)$. The value s_1 controls the accuracy of the estimate and the value s_2 controls the confidence of the estimate. Independent instances can be generated by using independent random seeds for generating the four-wise independent random variables. Note that ξ_q is not explicitly stored as part of the sketches, but is computed during query processing using the random seed for each sketch. We now state the following theorem.

Theorem 1 *Suppose S denotes a stream of one-dimensional values obtained from a stream of trees by using the pairing function $PF(\cdot)$. Let X be an AMS sketch for S. Let q be the one-dimensional mapping for query Q using $PF(\cdot)$. Then $COUNT_{ord}(Q)$ (*i.e., *f_q) over S can be estimated with a relative error of at most ϵ with probability at least $1-\delta$ using $s_1 \times s_2$ instances of X, where $s_1 = \frac{8SJ(S)}{\epsilon^2 f_q^2}$ and $s_2 = 2lg\frac{1}{\delta}$.*

Proof. Provided in the extended version [25]. □

3.2. Estimating the Frequency of a Set of Distinct Tree Patterns

For a given set of distinct tree patterns $\{Q_1, Q_2, \cdots, Q_t\}$, `SketchTree` can estimate their total frequency $\sum_{j=1}^{t} COUNT_{ord}(Q_j)$. We shall first construct an unbiased estimator and then compute an upper bound for its variance. For $1 \leq j \leq t$, let q_j ($\in dom(S)$) denote the one-dimensional mapping of a tree pattern Q_j. Due to the property of the Prüfer sequence transformation and pairing function $PF(\cdot)$, each q_j is distinct. We show that $X \cdot (\sum_{j=1}^{t} \xi_{q_j})$ is an unbiased estimator of the total frequency $\sum_{j=1}^{t} f_{q_j}$.

$$X \cdot \left(\sum_{j=1}^{t} \xi_{q_j}\right) = \sum_{j=1}^{t} \xi_{q_j}^2 f_{q_j} + \sum_{j=1}^{t} \sum_{1 \leq i \leq n, i \neq q_j} \xi_{q_j} \xi_i f_{q_j} f_i$$

$$E\left(X \cdot (\sum_{j=1}^{t} \xi_{q_j})\right) = \sum_{j=1}^{t} f_{q_j} \qquad (2)$$

By evaluating the expression for variance and applying Cauchy-Schwarz Inequality [22], we obtain the following result. In the interest of space, the details of the evaluation are provided in the extended version of this paper [25].

$$Var\left[X \cdot (\sum_{j=1}^{t} \xi_{q_j})\right] \leq 2(t-1) \cdot SJ(S) \qquad (3)$$

Now the following theorem can be stated in a way similar to Theorem 1.

Theorem 2 *Let X be an AMS sketch for a stream of one-dimensional values S, obtained from a stream of trees by using the pairing function $PF(\cdot)$. Given a set of distinct query patterns $\{Q_1, Q_2, \cdots, Q_t\}$, let q_j ($\in dom(S)$) be the one-dimensional mapping of Q_j using $PF(\cdot)$. The total frequency $\sum_{j=1}^{t} COUNT_{ord}(Q_j)$ can be estimated with a relative error of at most ϵ with probability at least $1-\delta$ using $s_1 \times s_2$ instances of X, where $s_1 = \frac{16(t-1)SJ(S)}{\epsilon^2 (\sum_{j=1}^{t} f_{q_j})^2}$ and $s_2 = 2lg\frac{1}{\delta}$.*

Proof. Similar to the proof of Theorem 1. □

Alternatively, the total frequency could be estimated by first estimating the frequency of each pattern separately and then computing the sum. A relative error of ϵ can be guaranteed if each individual estimation guarantees a relative error of $\frac{\epsilon}{t}$. Thus if $s_1 = \frac{8t^2 SJ(S)}{\epsilon^2 (min(f_{q_1}, \cdots, f_{q_j}))^2}$ then a relative error of ϵ can be guaranteed. From Theorem 2, it is clearly evident that using our proposed technique (Equation (2)) requires a smaller value for s_1 to guarantee a certain level of accuracy.

Algorithm 1: Update Process in `SketchTree`

Input: (T, k, s_1, s_2): T - input tree, k - maximum tree pattern size, s_1, s_2 - # of iid instances
Output: none

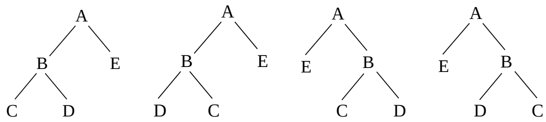
```
procedure SketchTreeUpdate(T, k, s1, s2)
for each tree pattern T_p generated by EnumTree(T,k) do
    compute LPS(T_p) and NPS(T_p)
    compute t_p ← PF(LPS(T_p).NPS(T_p))
    for i = 1 to s2 do
        for j = 1 to s1 do
            compute ξ_{t_p} using the random seed for sketch X_ij
            and add it to X_ij
        endfor
    endfor
endfor
```

Algorithm 2: Query Processing using `SketchTree`

Input: (Q_{list}, s_1, s_2): Q_{list} - list of query patterns s_1, s_2 - # of iid instances
Output: count estimate for Q_{list}

```
procedure SketchTreeEstimate(Q_list, s1, s2)
for each query pattern Q_l in Q_list do
    compute LPS(Q_l) and NPS(Q_l)
    compute q_l ← PF(LPS(Q_l).NPS(Q_l))
endfor
for i = 1 to s2 do
    for j = 1 to s1 do
        ξ ← 0
        for each Q_l in Q_list do
            compute ξ_{q_l} using the random seed of X_ij
            ξ ← ξ + ξ_{q_l}
        endfor
        Z_j ← ξ · X_ij
    endfor
    Y_i ← (Z_1 + ... + Z_{s1}) / s1
endfor
return median{Y_1, Y_2, ..., Y_{s2}}
```

The steps involved in `SketchTree` to update the synopsis data structure, when a new tree arrives in the input stream, is shown in Algorithm 1. Note that the tree-to-sequence transformation and applying the pairing function take linear time in the size of the tree pattern. The steps involved during query processing are shown in Algorithm 2. The query pattern is also mapped to a one-dimensional value. The ξ random variables are generated for each sketch, and the standard boosting technique is applied to compute an estimate.



(a) Q_1 (b) Q_2 (c) Q_3 (d) Q_4

Figure 4. Ordered Tree Patterns of Q

3.3. Unordered Tree Pattern Counts

`SketchTree` supports counting unordered tree pattern matches with provable guarantees on the approximation errors. Consider an unordered tree pattern Q with four different ordered tree pattern arrangements Q_1, Q_2, Q_3 and Q_4 as shown in Figure 4. In order to estimate $COUNT(Q)$, we use the results obtained in Section 3.2. Let q_1, q_2, q_3 and q_4 be the one-dimensional mappings of the patterns Q_1, Q_2, Q_3 and Q_4 respectively. Since the tree patterns are distinct, q_1, q_2, q_3 and q_4 are distinct integer values. From Equation (2), $COUNT(Q)$ can be computed by $\sum_{j=1}^{4} COUNT_{ord}(Q_j)$, which in turn can be estimated using an unbiased estimator $Y = X \cdot (\xi_{q_1} + \xi_{q_2} + \xi_{q_3} + \xi_{q_4})$.

4. Generalization of Count Queries

In this section, we generalize the class of queries that can be estimated by `SketchTree`. The benefit of `SketchTree` is that probabilistic guarantees on the quality of approximation can be provided. We provide some use cases of `SketchTree` in this section for applications that process tree structured data.

The `SketchTree` algorithm can estimate a class of query expressions that are generated by the following grammar rules using the arithmetic operators '$+$', '$-$', and '$\times$'.

$$E \rightarrow E + E \mid E - E \mid E \times E$$
$$E \rightarrow COUNT_{ord}(Q)$$

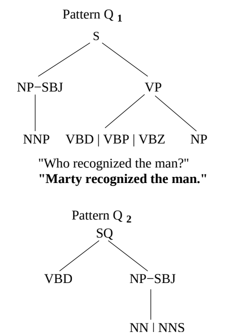
Note that $COUNT_{ord}(Q)$ is a terminal symbol for the rules. We assume that each terminal symbol in the query expression is distinct. For example, the tree patterns Q_1, Q_2 and Q_3 being counted in an expression '$COUNT_{ord}(Q_1) + COUNT_{ord}(Q_2) + COUNT_{ord}(Q_3)$' are all distinct.

For any valid query expression E, `SketchTree` constructs an unbiased estimator for E using the following procedure. Each $COUNT_{ord}(Q_i)$ in E is replaced by $\xi_i X$ to yield a new expression $E^{'}$. The expression $E^{'}$ is a polynomial of X. Each term in $E^{'}$ is divided by the factorial of the power of X in that term to yield $E^{''}$. We claim that $E^{''}$ is an unbiased estimator for the query expression E [25]. Note that for higher powers of X in $E^{''}$, four-wise independent ξ variables may not be sufficient. In general, we would need *k-wise* independent ξ random variables, where $k > 4$. A general technique has been proposed for generating *k-wise* independent binary random variables [1].

In the linguistic research community, language treebanks are commonly used, because treebanks provide a syntactic structure for text data by breaking them into syntactic units such as noun clauses, verbs, adjectives and so on. Treebanks can be modeled as ordered labeled trees and can be represented in XML (Figure 5(a)). A linguist could experimentally verify different hypotheses in a language by analyzing its treebanks [29].

```
.
.
<S>
  <NP-SBJ>
    <NNP>Marty</NNP>
  </NP-SBJ>
  <VP>
    <VBD>recognized</VBD>
    <NP>
      <DT>the</DT>
      <NN>man</NN>
    </NP>
  </VP>
</S>
.
.
```

(a) A snippet of Treebank in XML



(b) Query Patterns

Figure 5. Treebank Processing

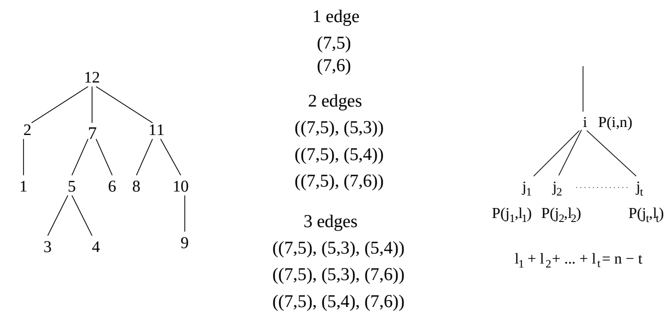
Due to space constraints, we present only two use cases and refer readers to the extended version of this paper [25] for more.

Example 3 *A language such as English uses the* `subject-verb-object` *word order for a sentence. However, a language that supports free word order uses any six permutations of* `subject`, `object` *and* `verb` *and each permutation is grammatically correct (*e.g., *German, Hindi). For example, a linguist could verify the following hypothesis experimentally: "Does a language L support free word order and if so to what extent?" Such an experimental validation requires counting tree patterns with nodes corresponding to* `subject`, `object` *and* `verb` *arranged differently.* `SketchTree` *can provide tight approximations to the actual counts quickly for large treebanks. Moreover, approximately counting syntactic structures can be useful for clustering large amounts of treebank data based on different language properties.*

Example 4 *Another common use of treebanks is in question answering systems [23]. A linguist may want to know how many sentences in the data denote the answer to a 'who' or 'how' or 'what' or 'when' question. Consider the query pattern Q_1 in Figure 5(b). The operator '$|$' in the query denotes a boolean OR. There is a match in the data (Figure 5(a)). This match is the answer to a 'who' question [23]: "Who recognized the man?" The number of such questions (*e.g., *who, what, how) that can be constructed from the data can be quickly estimated with a desired level of accuracy by* `SketchTree` *when the dataset is large. For example, the number of 'who' question can be estimated as follows. Query Q_1 can be represented by three distinct queries each containing all the nodes of Q_1 expect the node with the OR predicate 'VBD$|$VBP$|$VBZ'. The left child of VP in each query contains one operand of the OR operator. Let Q_{11}, Q_{12} and Q_{13} denote the three query patterns. Then an estimate of $\sum_{j=1}^{3} COUNT_{ord}(Q_{1j})$ computed by* `SketchTree` *in a single pass over the treebank data is an approximate answer for the total number of 'who' questions.*

5. Optimizing Processing Cost and Memory Utilization

In this section, we first describe an algorithm for enumerating all ordered tree patterns with at most k edges from an input



(a) Tree T (b) Trees rooted at '7' (c) Picking edges

Figure 6. $EnumTree$

data tree. We then present two strategies to improve the memory utilization of `SketchTree`, namely, (1) *tracking top-k frequent tree patterns* and (2) *virtual streams*. These two strategies can be combined together or can be applied separately. Limiting the number of instances of AMS sketches reduces the sketch update cost during stream processing.

5.1. Tree Pattern Generation by $EnumTree$

It is essential that `SketchTree` generates tree patterns from a stream of trees efficiently. We propose an intuitive algorithm called $EnumTree$ to enumerate all ordered tree patterns in an ordered labeled tree with at most k edges. $EnumTree$ constructs larger tree patterns from smaller tree patterns and uses a *memoization technique* to avoid repeated computation of the same patterns.

Let i denote a unique identifier of a node in the input tree. Let $P(i,j)$ denote the set of tree patterns in the input tree rooted at node i with j edges each. Each tree pattern can be represented by a set of edges where each edge is denoted by node id pairs. If there are no qualifying tree patterns with j edges rooted at i, then $P(i,j) = \varnothing$. In addition, let $P(i,0) = \perp$.

The $EnumTree$ algorithm generates patterns rooted at each node in a data tree by visiting the nodes in postorder. To compute $P(i,j)$, $EnumTree$ first picks a set of child edges of i and then picks the remaining edges from its descendants. Consider a data tree T in Figure 6(a), where nodes are numbered in postorder. Suppose $EnumTree$ is currently at node 7 to compute $P(7,3)$. There are three choices for edge selection: $(7,5), (7,6)$ and $((7,5),(7,6))$. If $(7,5)$ is picked, then $EnumTree$ next computes $P(5,2)$, which returns edges $\{(5,3),(5,4)\}$. Similarly, if $(7,6)$ is picked, $EnumTree$ computes $P(6,2)$, which is $\varnothing$. As a result, a tree pattern $\{(7,5),(5,3),(5,4)\}$ is generated. Lastly, if $((7,5),(7,6))$ is picked, then $EnumTree$ has to choose one more edge from its descendants. There are two possibilities: $\{P(5,1),P(6,0)\}$ and $\{P(5,0),P(6,1)\}$. $P(5,1)$ contains edges $(5,3)$ and $(5,4)$, while $P(6,1)$ is $\varnothing$. As a result, tree patterns $\{((7,5),(7,6),(5,3)),((7,5),(7,6),(5,4))\}$ are generated. In a similar way, $P(7,2)$ and $P(7,1)$ are computed. The trees rooted at node 7 are shown in Figure 6(b).

As shown in Figure 6(c), in general, to compute $P(i,n)$, if t child edges are chosen, then $P(j_1,l_1)$, $P(j_2,l_2)$, $\cdots$, $P(j_t,l_t)$ are computed $\forall l_1,\cdots,l_t \geq 0$ such that $l_1+\cdots+l_t = n-t$. To compute $P(i,n)$, the cartesian product

$$C \quad = \quad P(j_1,l_1) \times P(j_2,l_2) \times \cdots \times P(j_t,l_t) \qquad (4)$$

is first computed. Each result in C along with the edges (i,j_1), (i,j_2), $\cdots$, and (i,j_t) denotes a tree pattern of size n rooted at i. Note that if any $P(\cdot) = \perp$ in Equation (4), then it is not included in the cartesian product. Also if any $P(\cdot) = \varnothing$ in the equation, then $C = \varnothing$ and $P(i,n) = \varnothing$.

It can be observed that due to the recursive nature of our algorithm, $P(i,j)$ may be invoked many times. To avoid repeated computations, $EnumTree$ stores each solution set $P(i,j)$. If n is the maximum number of allowed edges for a tree pattern generated by $EnumTree$, then only those solution sets with $j < n$ need to be stored. We provide an empirical evaluation of the effectiveness of $EnumTree$ in Section 7.

5.2. Tracking Top-k Frequent Tree Patterns

We present an intuitive strategy to reduce the memory requirement of `SketchTree` by tracking the top-k frequent tree patterns in a stream. Theorems 1 and 2 show that the memory requirement of `SketchTree` depends on the self-join size of the input stream. Thus by reducing the self-join size, we can improve the accuracy of the estimate for a given amount of memory.

A key benefit of using AMS sketches is that the process of deleting values from a stream is straightforward. Consider a stream of integers S that is sketched by X. A value t can be deleted from this stream by subtracting ξ_t from X. Furthermore, m instances of t can be deleted from S by subtracting $m\xi_t$ from X. Suppose the values in S have a *skewed distribution*. Then, the deletion of top-k frequent values from S can potentially result in substantial reduction in the self-join size of S.

At any instant of time, $E(\xi_t \cdot X)$ is an unbiased estimator of the frequency of t in S so far. (See Section 3.1.) The key intuition for estimating the frequency of t will be clear from the following analysis. Using Markov's Inequality [22], we obtain the following equation.

$$P(\xi_t \cdot X \geq r) \quad \leq \quad \frac{E(\xi_t \cdot X)}{r} \qquad (5)$$

If r is large and the actual frequency of t, $E(\xi_t \cdot X)$, is small, then the probability of the estimated frequency of t being larger than r is very small. Essentially, during stream processing, the probability that a low frequency value is (incorrectly) estimated as *frequent* is very small. Equation (5) forms the basis of our memory reduction strategy.

The data structures maintained by `SketchTree` include a min-heap H and a list L, each of size k. Thus at most k most frequent values (mappings of tree patterns) can be tracked. H stores the estimated frequencies of these frequent tree patterns that are present in L. Before the start of stream processing, both H and L are empty.

Algorithm 3 describes the steps involved during stream processing. Let f_i be a frequency estimate of i. At any point during top-k processing, the following `delete` condition holds. *If frequent value i is present in L, then f_i instances of i have been deleted from the stream.* Since the self-join size of the stream affects the accuracy of the estimates computed during

Algorithm 3: Tracking Top-k Frequent Tree Patterns

Input: (t, s_1, s_2): t - 1D value; s_1, s_2 - # of iid instances
Output: none

procedure `ComputeTopK`(t, s_1, s_2)
1: **if** *t is present in L* **then**
2: Let f_t be the frequency of t stored in H
3: **for** *i = 1 to* s_2 **do**
4: **for** *j = 1 to* s_1 **do**
5: Compute ξ_t using the random seed for X_{ij}
6: Add $\xi_t \cdot f_t$ to X_{ij}
7: Delete t and f_t from L and H respectively
endfor
endfor
endif
8: Compute $estFreq_t$ using $s_1 \times s_2$ instances of sketch X
9: **if** $estFreq_t > 0$ *and* $estFreq_t >$ *Root(H)* **then**
10: **if** *HeapSize*$(H) = k$ **then**
11: Let f_r be the root of heap H and r be its corresponding element in L
12: Add $\xi_r \cdot f_r$ back to each X_{ij}
13: Delete root of H and delete r from L
endif
14: Insert $estFreq_t$ into H and add t to L
15: **for** *i = 1 to* s_2 **do**
16: **for** *j = 1 to* s_1 **do**
17: Compute ξ_t using the random seed for X_{ij}
18: Delete $\xi_t \cdot estFreq_t$ from X_{ij}
endfor
endfor
endif

top-k processing, the `delete` condition results in low estimation errors. In Algorithm 3, if input t is present in L, then f_t instances of t are added back to the stream and all the instances of the sketches are updated appropriately (Lines 1 through 7). Next the frequency of t is again estimated using $\xi_t \cdot X$ by using $s_1 \times s_2$ instances of X (Line 8). If the estimated frequency $estFreq_t$ is positive and is greater than the minimum frequency in the heap H, H and L are updated (Lines 9 through 14). If H is full, then all the instances of the frequent value corresponding to the root are added back to the sketches (Line 12). The root node in H is deleted and the corresponding frequent value is deleted from L. Finally, t and $estFreq_t$ are inserted into L and H respectively (Line 14). Then $estFreq_t$ instances of t are deleted from the stream and all the sketches are updated (Lines 15 through 18). Note that the `delete` condition still holds.

Note that Algorithm 3 is invoked with t_p, s_1 and s_2 as input arguments, after all the $s_1 \times s_2$ sketches have been updated in Algorithm 1 (Lines 4 through 6). If invoking top-k processing for every tree pattern generated by $EnumTree(\cdot)$ is infeasible for an application, then top-k processing could be invoked with a probability p for each tree pattern. In Algorithm 3, a sorted data structure such as a `map` container may be used for L to speed up insert, delete and search operations.[1]

A few modifications to the query processing algorithm (Algorithm 2) of `SketchTree` are necessary. The basic idea is to temporarily add the deleted instances of frequent values in list L, that are also present in the query list, to the sketches. Let f_{q_l} be the frequency of q_l (mapping of query $Q_l \in Q_{list}$) that is present in L. For each sketch X_{ij}, compute $d = \sum_{q_l \in L} \xi_{q_l} f_{q_l}$. Line 10 in Algorithm 2 is replaced by $Z_j \leftarrow \xi \cdot (X_{ij} + d)$.

5.3. Virtual Streams

Another memory reduction technique that `SketchTree` uses, is to split a single stream of one-dimensional integer values into a set of disjoint *virtual streams*. As a result, each virtual stream has a smaller self-join size as compared to the original stream. When a new value appears in the original stream, the value is inserted into one of the virtual streams. This approach is similar to using a set of buckets in COUNT SKETCHES [8].

Let p be a prime that denotes the number of virtual streams S_0, S_1, ..., S_{p-1} for the stream S. Note that $dom(S) = \bigcup_{i=0}^{p-1} dom(S_i)$. For each one-dimensional value t that appears in the original stream, we compute residue $r = t$ mod p.[2] Now the value t is inserted into the r^{th} virtual stream S_r. `SketchTree` now maintains AMS sketches for each virtual stream. Let X_i denote an AMS sketch for a virtual stream S_i. In order to estimate COUNT(Q), the one-dimensional mapping q for a query Q is used to compute the residue $r_q = q$ mod p. The sketch X_{r_q} of the virtual stream S_{r_q} is used for computing an approximate answer for COUNT(Q).

The sketches X_0, X_1, ..., X_{p-1} can share the same random seed to generate four-wise (or k-wise) independent ξ variables. So an AMS sketch for say $S_i \cup S_j$ is simply the addition $X_i + X_j$. `SketchTree` can estimate any query expression (Section 4) by first computing the addition of all the relevant sketches for the query trees in the expression. The sum of the values in these sketches is used during query processing. The top-k strategy can be combined with virtual streams. In such a case, `SketchTree` would maintain a separate top-k data structure for each virtual stream.

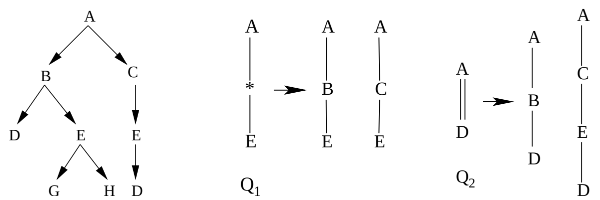
6. Extensions

6.1 Alternate Mapping Function

We have so far assumed that the pairing function $PF(\cdot)$ maps a given LPS and NPS pair to an integer. As the number of elements in these sequences increases and the element values grow, the range of $PF(\cdot)$ grows too. In such cases, a 32-bit word (or a 64-bit word) may not be sufficient to store the integers. We extend `SketchTree` by adopting Rabin's fingerprinting technique [6] as a mapping function instead of $PF(\cdot)$. In Rabin's work, an irreducible polynomial of large degree is chosen uniformly at random. Let p_{irr} denote such a polynomial of degree 31 (assuming the use of 32-bit integers). A given LPS and NPS pair can be concatenated, and the whole sequence can be treated as a long bit string representing the coefficients of a polynomial with coefficients 0 or 1. Let p denote such a polynomial. The residue polynomial $r = p$ mod p_{irr} is considered a one dimensional point mapping for the LPS and NPS. Note that r has a smaller degree than that of p_{irr} and can be stored as a 32 bit integer.

[1] The `map` container is available in C++ Standard Template Library.

[2] Universal hash families can also be used [22].



(a) Structural summary (b) Wildcard '*' (c) Axis '//'

Figure 7. Processing Queries with '*' and '//'

We have also assumed so far that each node label X in the data is mapped to a unique number using $hash(X)$. (See Section 2.) This mapping can be done in an online fashion too. The node labels can be treated as bit strings and residue polynomials can be computed in the same way as described earlier.

It should be noted that the mapping scheme using irreducible polynomials can lead to collisions. However, the probability of collisions can be made very low by using irreducible polynomials of appropriate degrees [6]. For our experiments, we chose irreducible polynomials of degree 31.

6.2 Extending the Query Semantics

In the streaming scenario studied in this paper, we have assumed that there exists no structural summary such as a schema on the data trees. However, if a structural summary exists or can be constructed online using limited space, then `SketchTree` can be extended to process queries that contain ancestor-descendant relationship between nodes ('//' in XPath) and wildcard nodes ('*' in XPath). Our approach is similar to that of XSKETCHES [24] in the sense that the original queries are mapped to set of query patterns with only parent-child edges. The total frequency of this set of query patterns is equal to the frequency of the original pattern.

Suppose a structural summary of data is available as shown in Figure 7(a). In order to process query Q_1 shown in Figure 7(b), the structural summary can be used and '*' can be resolved into two labels B and C. Thus $COUNT_{ord}(Q_1)$ is the sum of the frequencies of two distinct patterns shown in Figure 7(b). Similarly, to process a query Q_2 shown in Figure 7(c), the structural summary can be used and '//' can be resolved to yield two distinct patterns shown alongside. Thus $COUNT_{ord}(Q_2)$ is the sum of the frequencies of these two distinct patterns. Recall that in Section 3.2, we show how `SketchTree` can estimate the frequency of any set of distinct tree patterns. Note that we assume that the resulting tree patterns are within size k each where k is the size of the largest tree pattern generated by $EnumTree$. Otherwise, this simple sum of frequencies technique cannot be applied. As part of future work, we would like to address issues such as choosing the right value for k, and counting tree patterns of size larger than k.

7. Experimental Results

In this section, we present the experimental evaluation of `SketchTree` done with real datasets. We computed the average relative errors for $COUNT_{ord}(\cdot)$ queries on workloads with varying query selectivities. We observed that using a limited amount of memory, `SketchTree` could estimate tree pattern counts within 10-15% relative error. In addition, we observed that the cost of processing data trees grew almost linearly with the total number of tree patterns generated by $EnumTree$.

Dataset	# of Trees	Maximum Tree Pattern Size (k)	# of Distinct Tree Patterns
TREEBANK	28,699	6	7,041,113
DBLP	98,061	4	11,301,512

Table 1. Datasets

7.1. Experimental Setup

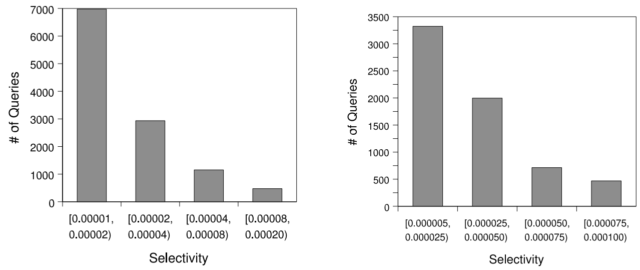
The `SketchTree` was developed in C++ with the GNU Scientific Library (GSL) for generating pseudo random numbers. We ran all our experiments on 2.4GHz Pentium IV processor with 1 GB RAM running Red Hat Linux 9.0.

7.2. Data Sets

We experimented with two real datasets (a) TREEBANK and (b) DBLP [28]. Each dataset was originally a single large XML document. A forest of trees were created by removing the root tag of the document, and the trees were processed in a single pass. The trees in TREEBANK were narrow and deep with recursive element names. The trees in DBLP were shallow and bushy. Table 1 summarizes the total number of trees processed, the maximum size of a tree pattern that was generated by $EnumTree$, and the total number of distinct ordered tree patterns in each dataset. Recall that a deterministic counting approach would require one counter for each distinct tree pattern, which would amount to more than 7 million and 11 million counters for TREEBANK and DBLP datasets, respectively.

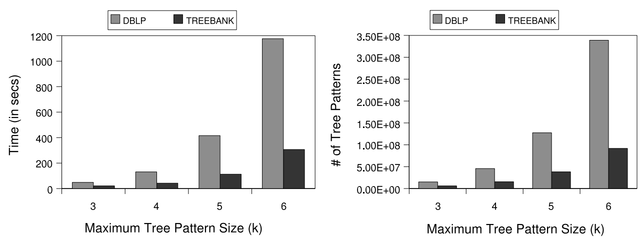
7.3. Query Workload

For each dataset, a query workload was generated by selecting ordered tree patterns from it with different selectivities. Figure 8(a) shows the workload for TREEBANK with the number of queries at each selectivity range. The size (*i.e.*, number of edges) of each query pattern ranged from 1 to 6. For TREEBANK, since its value data were encrypted, the queries had only element names. All the queries were in the selectivity range $[0.00001, 0.00020)$, and the actual counts of these queries ranged in the interval $[872, 18256]$. Figure 8(b) shows the workload for DBLP with the number of queries at each selectivity range. The size of each query pattern ranged from 1 to 4. For DBLP, the queries had element names as well as values (CDATA). All the queries were in the selectivity range $[0.000005, 0.0001)$, and the actual counts of these queries ranged in the interval $[206, 4547]$.



(a) TREEBANK (b) DBLP

Figure 8. Query Workload



(a) Time for Generation (b) Number of Tree Patterns

Figure 9. Evaluation of `EnumTree`

7.4. Tree Pattern Generation Cost

A core component of `SketchTree` is the process of generating tree patterns from data trees. In Section 5.1, we have proposed an algorithm, $EnumTree$, that given a tree T and a value k, enumerates all the ordered tree patterns in T, each with at most k edges. We evaluated the performance of $EnumTree$ by measuring the total wall clock time for processing all the trees in DBLP and TREEBANK for different values of k. This time included the time to generate the patterns, to transform them into sequences, and to compute their one-dimensional mappings using Rabin's technique.

In Figure 9(a), the total time taken by $EnumTree$ to process all the trees in the stream is plotted for both TREEBANK and DBLP for different values of k. In Figure 9(b), the total number of ordered tree patterns generated by $EnumTree$ are plotted for different values of k. The similarity between the two plots shows that the the time taken by $EnumTree$ grows almost linearly with the number of tree patterns that are generated, which attests the effectiveness of $EnumTree$ for generating tree patterns. Note that the number of tree patterns generated for DBLP was larger than that for TREEBANK. The reason was that since DBLP had a larger fanout for the tree nodes, there were more choices for picking child edges during enumeration.

7.5. Quality of Answers and Memory Usage

The quality of approximate answers for $COUNT_{ord}(\cdot)$ queries can be measured by computing the standard relative error $\frac{|approx - actual|}{actual}$. Note that an approximate count can be negative. In such cases, we use a sanity bound for the approximate count by $approx = 0.1 \times actual$.

As more memory is allocated for the synopses in `SketchTree`, we expect the relative error to decrease. We evaluated the effectiveness of `SketchTree` by increasing the number of instances of the sketches by varying the value of s_1 in increments of 25 with s_2 being fixed at 7.[3] In addition, the top-k size (# of frequent patterns to track) was increased in increments of 50. Note that the number of virtual streams (Section 5.3) was fixed at 229 for all the experiments. (An increase in this number would reduce the self-join size of the streams and provide better accuracy as expected.) For each query, we computed the average relative error over 5 runs for a given value of s_1 and and top-k size. The total memory allocated for the synopses in `SketchTree` is equal to sum of the memory required for $s_1 \times s_2$ iid instances of AMS sketches, top-k data structures and independent random seeds required for constructing four-wise independent binary random variables.

The plots in Figure 10 show the average of the average relative error for the set of queries in each selectivity range. Note that in this case, `SketchTree` estimated the counts of single tree pattern queries. In the interest of space, we report additional experimental results for estimating query expressions (*e.g.*, sum, product of tree pattern counts), described in Section 4, in the extended version of this paper [25]. Based on the theorems stated in Section 3, we expect the average relative error to decrease as a query becomes less selective. In addition, the accuracy is expected to improve as the top-k size is increased, since more high frequency values would be deleted from the virtual streams resulting in a lower self-join size.

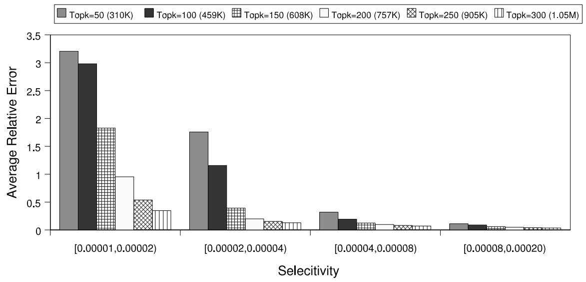
7.6. TREEBANK

For TREEBANK, the average relative errors were computed for the query workload shown in Figure 8(a). The results are shown in Figures 10(a) and 10(b). The total memory allocated for the synopses and top-k data structures is also shown in these plots.

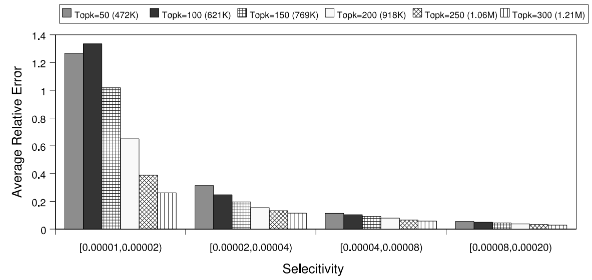
Figure 10(a) shows the average relative errors for the case when s_1 was 25. The total memory allocated ranged from 316 KB to 1.05MB. We observed that with increase in the top-k size, the average relative error dropped steadily. This was because `SketchTree` removed high frequency values (integer mappings of frequent tree patterns) from the sketches, thereby reducing the self-join sizes of the virtual streams during query processing. For example, the average relative error, for the selectivity range $[0.00002, 0.00004)$, dropped from 1.76 (176%) to 0.15 (15%) when the top-k size was increased from 50 to 250. For the selectivity range $[0.00004, 0.00008)$, the relative error was below 12% for top-k size from 150 onwards.

Figure 10(b) shows the average relative errors for the case when s_1 was 50. The total memory allocated to `SketchTree` ranged from 472 KB to 1.21 MB. The average relative error dropped steadily with increase in the top-k size as before. For example, the average relative error for the selectivity range $[0.00001, 0.00002)$ dropped from 1.27 (127%) to 0.39 (39%) when the Topk size was increased from 50 to 300. For the selectivity ranges $[0.00004, 0.00008)$ and $[0.00008, 0.00020)$ the

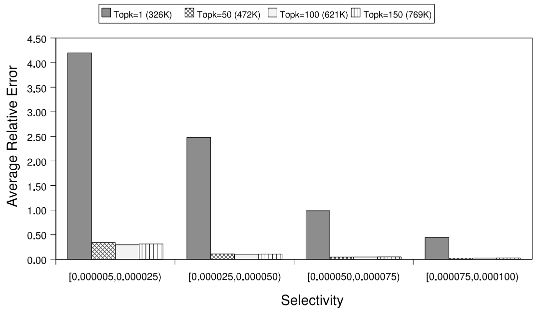
[3] We computed the value of s_2 for $\delta = 0.1$ using Theorem 1.



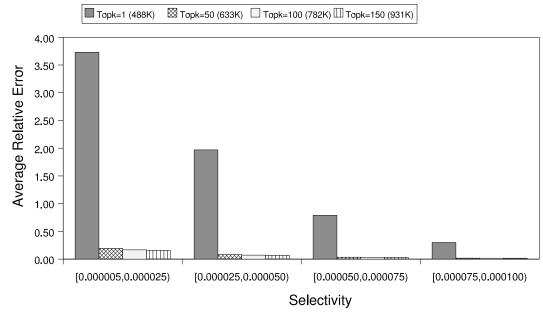
(a) TREEBANK ($s_1 = 25$, $s_2 = 7$)



(b) TREEBANK ($s_1 = 50$, $s_2 = 7$)



(c) DBLP ($s_1 = 50$, $s_2 = 7$)



(d) DBLP ($s_1 = 75$, $s_2 = 7$)

Figure 10. Evaluation of `SketchTree`

relative errors were below 12% for Topk sizes from 50 to 300.

We conclude that tracking frequent tree patterns and deleting them from the sketches is an effective way of boosting the accuracy of estimates computed by `SketchTree`. We also observed that by increasing the value of s_1, the relative error dropped significantly for the same value of top-k size. This is consistent with the theoretical analyses. However, the stream processing time increased when s_1 was increased. We computed the ratio of total stream processing time when $s_1 = 50$ to the processing time when $s_1 = 25$. We observed that the processing cost increased by a factor of 2.3 when s_1 was doubled for different top-k sizes. Interestingly, when the top-k size was increased for a fixed value of s_1, the increase in the processing cost was marginal. For example, when top-k value was increased from 50 to 300, the processing cost increased by only 5.4% and 4.0% for $s_1 = 25$ and $s_1 = 50$ respectively.

However, the improvement in accuracy may not as by increasing the top-k size, is not a clear winner always. Since the accuracy of estimating a frequent value is proportional to the actual value, tracking more and more values will result in less accurate estimation for values that are less frequent. As a result, the quality of answers could degrade. In such cases, increasing the value of s_1 could be a better choice, provided if increase in processing time is feasible.

7.7. DBLP

For DBLP, the average relative errors were computed for the query workload in Figure 8(b). The results are shown in Figures 10(c) and 10(d). The total memory allocated for the synopses and top-k data structures is also shown in these plots.

Figure 10(c) shows the average relative errors for the case when s_1 was 50. The total memory allocated ranged from 326 KB to 769 KB. We observed a drastic improvement in accuracy when the top-k size was increased from 1 to 50. On the contrary, we observed a more gradual improvement in accuracy for TREEBANK. This is because the distribution of tree patterns in DBLP had higher degree of skew than the tree patterns in TREEBANK. As a result, for DBLP, deleting fewer frequent patterns from the virtual streams were sufficient to compute estimates with good accuracy. For example, the average relative error for the queries in the selectivity range $[0.000025, 0.000050)$, dropped from 2.48 (248%) to 0.11 (11%) when the top-k size was increased from 1 to 50. With further increase in top-k size, the improvement in accuracy was marginal. For the selectivity ranges $[0.000050, 0.000075)$ and $[0.000075, 0.0001)$, the relative errors were under 5% for top-k size of 50.

Figure 10(d) shows the results for the case when s_1 was 75. The total memory allocated ranged from 488 KB to 931 KB. The average relative error dropped drastically as before when the top-k size was increased from 1 to 50 due to high skew in the tree pattern distribution. For example, the average relative error for queries in the selectivity range $[0.000005, 0.000025)$ dropped from 3.75 (375%) to 0.19 (19%) when the top-k size was increased from 1 to 50. (The relative errors can be further reduced by increasing the value of s_1.) For the remaining selectivity ranges, relative error under 8% was achieved using `SketchTree` for top-k size of 50.

As before, our experiments show that deleting frequent patterns from the sketches is an effective way of improving the accuracy of of `SketchTree` estimates. As expected, with increase in the value of s_1, the relative errors dropped significantly for the same top-k size. However, the stream processing time increased. We computed the ratio of total stream processing time when $s_1 = 75$ to the processing time when $s_1 = 50$.

We observed that the processing cost increased by a factor of about 1.6 when s_1 was increased from 50 to 75 for different top-k sizes. However when top-k size was increased from 1 to 150 by fixing the value of s_1, the the processing cost increased by only 8.2% and 9.8% for $s_1 = 50$ and $s_2 = 75$ respectively.

8. Conclusion and Future Work

In this paper, we have addressed the problem of counting tree patterns on streaming labeled trees (*e.g.*, XML documents). We propose a new algorithm called `SketchTree` that constructs a synopsis of the stream using AMS sketches and provides an approximate answer for the number of occurrences of any tree pattern. `SketchTree` can estimate a class of counting queries including unordered tree pattern counts. We provide theoretical analyses to show that our algorithm has provably strong guarantees on the error bound for different types of queries. We show that the memory requirement of `SketchTree` can be further reduced by keeping track of high frequency tree patterns. We also present empirical results to demonstrate that `SketchTree` can estimate tree pattern counts within relative errors of 10-15% using a limited amount of memory. `SketchTree` can be useful for tasks such as *selectivity estimation* over stored data, especially when the data is very large and multiple passes over the data is impractically expensive. As part of future work, we would like to compare `SketchTree` with techniques developed for selectivity estimation of twig queries such as XSKETCHES [24].

Acknowledgements

We would like to thank the anonymous reviewers for their valuable comments.

References

[1] N. Alon, L. Babai, and A. Itai. A Fast and Simple Randomized Parallel Algorithm for the Maximal Independent Set Problem. *Journal Of Algorithms*, 7:567–583, 1986.

[2] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *Proceedings of the 18th ACM Principles of Database Systems*, pages 10–20, 1999.

[3] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proceedings of the 28th ACM Symposium on Theory of Computing*, pages 20–29, 1996.

[4] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th VLDB Conference*, pages 53–64, Cairo, Egypt, Sept. 2000.

[5] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.0 second edition W3C recommendation. Technical Report REC-xml-20001006, World Wide Web Consortium, Oct. 2000.

[6] A. Z. Broder. Some Applications of Rabin's Fingerprinting Method. In *Sequences II: Methods in Communications, Security, and Computer Science*, 1993.

[7] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of the 2002 ACM-SIGMOD Conference*, Madison, Wisconsin, June 2002.

[8] M. Charikar, K. Chen, and M. Farach-Colton. Finding Frequent Items in Data Streams. In *Proceeding of the 29th International Colloquium on Automata Languages and Programming*, 2002.

[9] G. Cormode and S. Muthukrishnan. What's Hot and What's Not: Tracking Most Frequent Items Dynamically. In *Proceedings of the 2003 ACM-SIGMOD Conference*, San Diego, California, June 2003.

[10] Y. Diao, S. Rizvi, and M. Franklin. Towards an Internet-Scale XML Dissemination Service. In *Proceedings of the 30th VLDB Conference*, Toronto, Canada, Sept. 2004.

[11] A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the 2002 ACM-SIGMOD Conference*, Madison, Wisconsin, June 2002.

[12] M. Drmota. Combinatorics And Asymptotics On Trees. *Cubo Journal*, 6(2), 2004.

[13] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *Proceedings of the 28th VLDB Conference*, pages 454–465, Hong Kong, China, Aug. 2002.

[14] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proceedings of the 2001 ACM-SIGMOD Conference*, Santa Barbara, California, 2001.

[15] S. Homer and A. L. Selman. *Computability and Complexity Theory*. Springer, 2001.

[16] H. Jiang, W. Wang, H. Lu, and J. X. Yu. Holistic Twig Joins on Indexed XML Documents. In *Proceedings of the 29th VLDB Conference*, pages 273–284, Berlin, Germany, Sept. 2003.

[17] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. In *Proceedings of the 30th VLDB Conference*, pages 228–239, Toronto, Canada, Aug. 2004.

[18] J. Kwon, P. Rao, B. Moon, and S. Lee. FiST: Scalable XML Document Filtering by Sequencing Twig Patterns. In *Proceedings of the 31th VLDB Conference*, Trondheim, Norway, Aug. 2005.

[19] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th VLDB Conference*, pages 361–370, Rome, Italy, Sept. 2001.

[20] X. Li and G. Agrawal. Efficient Evaluation of XQuery over Streaming Data. In *Proceedings of the 31th VLDB Conference*, pages 265–276, Trondheim, Norway, Aug. 2005.

[21] G. Manku and R. Motwani. Approximate Frequency Counts over Data Streams. In *Proceedings of the 28th VLDB Conference*, Hong Kong, China, Aug. 2002.

[22] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 2000.

[23] K. Müller. Semi-Automatic Construction of a Question Treebank. In *Proceedings of the 4th International Conference on Language Resources and Evaluation*, Lisbon, Portugal, 2004.

[24] N. Polyzotis, M. Garofalakis, and Y. Ioannidis. Selectivity Estimation for XML Twigs. In *Proceedings of the 20th IEEE International Conference on Data Engineering*, Boston, MA, March 2004.

[25] P. Rao and B. Moon. Approximate Tree Pattern Counts over Streaming Labeled Trees. Technical Report TR 04-04, University of Arizona, Tucson, AZ 85721, Mar. 2004. http://www.cs.-arizona.edu/research/reports.html.

[26] P. Rao and B. Moon. PRIX: Indexing And Querying XML Using Prüfer Sequences. In *Proceedings of the 20th IEEE International Conference on Data Engineering*, Boston, MA, March 2004.

[27] A. C. Snoeren, K. Conley, and D. K. Gifford. Mesh-based Content Routing using XML. In *Proceedings of the 18th Symposium on Operating Systems Principles*, pages 160–173, Banff, Canada, Oct. 2001.

[28] UW XML Repository. Available from http://www.cs.washington.edu/research/xmldatasets.

[29] William D. Lewis. Personal communications. http://zimmer.-csufresno.edu/~wlewis/.